Compositional Specification of Software Architecture

John Penix
Automated Software Engineering Group
NASA Ames Research Center
M/S 269-3
Moffett Field, CA 94035
jpenix@ptolemy.arc.nasa.gov

Abstract

This paper describes our experience using parameterized algebraic specifications to model properties of software architectures. The goal is to model the decomposition of requirements *independent of the style* used to implement the architecture. We begin by providing an overview of the role of architecture specification in software development. We then describe how architecture specifications are build up from component and connector specifications and give an overview of insights gained from a case study used to validate the method.

1 Introduction

Existing formal models of software architecture are mainly concerned with formalizing specific architectural styles such as pipe-filter and client-server [1, 8, 12]. While architectural styles abstract away some implementation details, each still represents a highly reduced subset of the space of possible system designs. The reduction of the design space is what makes a style usable by human designers. However, the fact that the space is reduced indicates that the choice of an architectural style is an important design decision that should not be made prior to initial requirements specification. The alternatives for decomposing the system requirements should drive the selection of a specific architectural style.

In previous work, we described the role of declarative architecture specifications in system design [11]. Formally, an architecture specification is a parameterized specification where the parameters correspond to 'sockets' where components can be 'plugged-in' to the architecture. An architecture specification contains axioms that specify constraints on the component and system specifications. They may specify component behavior that is necessary to guarantee correct system-level behavior or define how variation in component behavior affects the behavior of the system. Additionally, you can state assumptions that the component can make about the environment provided by the architecture.

The system decomposition described by an architecture specification is implemented via an architecture schema written in a target programming language or architecture description language. The correctness of the implemented system requires that the constraints placed on the system by the specification axioms are guaranteed by the architecture schema. This can be verified based on a semantics of the target language [7, 11]. The key point is that verification of basic architectural properties is separated from the verification of specific system instantiations.

2 Compositional Architecture Specifications

A limitation of our previous work was that the specification of architecture constraints was monolithic. We have been investigating approaches to construct these architecture specifications from parts. One benefit of using a theory-based specification notation is that large specifications can be built up from smaller specifications using extension and parameterization [4, 6]. Therefore, we define theories for various kinds of connectors and bindings and they can be combined, extended and instantiated to create an architecture specification.

2.1 Component Specifications

We currently use a simple pre/post condition model of components. We are exploring more complex component (and connector) models to allow the approach to better support concurrency. However, the method of structuring specifications is independent of the component model. Therefore, the pre/post specifications are sufficient to explore the compositional specification approach.

2.2 Connector Specifications

A connection associates an output port of one component with an input port of another component. In general, the goal is to specify that the combined behavior of two components is always defined, i.e., every valid output at the source is a legal input at the destination. The specifics of this relationship depend upon the way that the components are connected.

For example, a Data Flow connector relates the output of one component directly to the input of another component. The verification condition for this type of connection is:

$$\forall x, w \; I_A(x) \wedge O_A(x, w) \Rightarrow I_B(w)$$

If this condition is true, then given a legal input to A, all valid outputs of A are legal inputs to B. We use this condition to create a generic data flow connection specification. The specification is parameterized on the interface specifications of the two components being connected. Specification have also been constructed for buffered, conditional and feedback connectors [9].

2.3 Interface Binding

To view a collection of interconnected components as an architecture, it must be associated with a single *system interface*. This is done by binding the inputs and outputs of the system interface with inputs and outputs of the subcomponent interfaces. There are several types of bindings that can be specified and used in an architecture specification. For example, the axiom:

$$\forall x \; I_{SYSTEM}(x) \Rightarrow I_{COMPONENT}(x)$$

specifies the correctness requirement when all of the system inputs are connected to all of the inputs of a single component. This specification is parameterized on the precondition of the system and the precondition of the component.

2.4 Architecture Specification

Architecture specifications are constructed by combining and extending component, connector and binding specifications. For example, Figure 1 shows a block diagram of an architecture for the Find problem. This architecture is represented by the specification diagram [6] in Figure 2. There are input bindings between a component specification and the problem specification for each of the input variables. The two components are connected by a data flow connector from the output of the first component to an input variable of the second component. The soundness axiom states the condition that must be true for the component behavior to properly implement the system level behavior. In this case, the soundness axiom has the form:

$$I(\langle a, k \rangle) \wedge O_A(a, c) \wedge O_B(\langle c, k \rangle, z)) \Rightarrow O(\langle a, k \rangle, z)$$

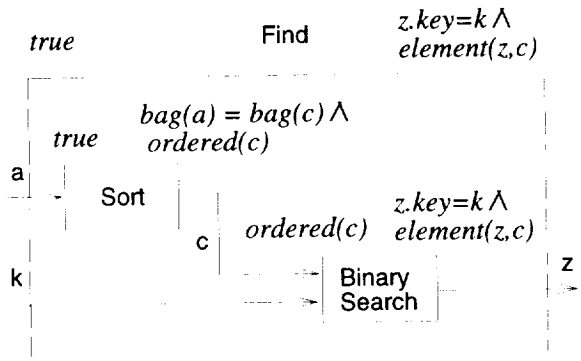

Figure 1: Example Find Architecture

where I and O are the system level specification and O_A and O_B are the postconditions of the two components. The generic problem and component specifications, together with the soundness axiom, represent the generic architecture theory, labeled `Partial Sequential` in the diagram. By plugging in the `Find` problem specification and the `Sort` and `BinSearch` component specifications, the specialized `Find Architecture` is created.

3 Case Study: KWIC

The Keyword In Context (KWIC) indexing system has been used as an example to present software architecture concepts. Shaw and Garlan's book [12] contains four solutions to the problem that have different architectures. The different architectures represent various combinations of choices regarding the tradeoffs on issues such as reusability, adaptability, performance. The goal of this case study was not to define a new architecture for KWIC, but to describe the existing KWIC architectures using compositional specifications. Details of the case study can be found elsewhere [9].

Traditionally, the KWIC problem is solved using the following four components: Input, Circular Shift, Alphabetize, and Output [12]. Our approach to specifying the KWIC architectures is to specify the interface and behavior of these four components, and then integrate them into the different architectures. The integration is non-trivial because, in the different architectures, the components have slightly different interfaces. We attempted to separate the specification of the core component functionality from the specification of component interaction. Then the core components were integrated using wrappers (specified as one-component architectures). The different interaction styles were captured by the connector constraints.

For example, a specification diagram for the KWIC data flow architecture is shown in Figure 3. The sub-diagram in the lower left shows how an incremental shift component is wrapped to create a Shift component with the correct

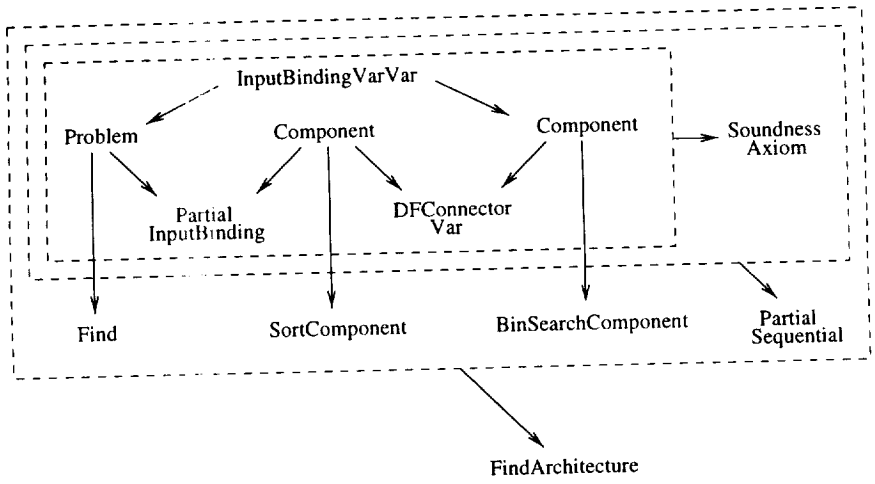

Figure 2: Specification Diagram for Find Architecture

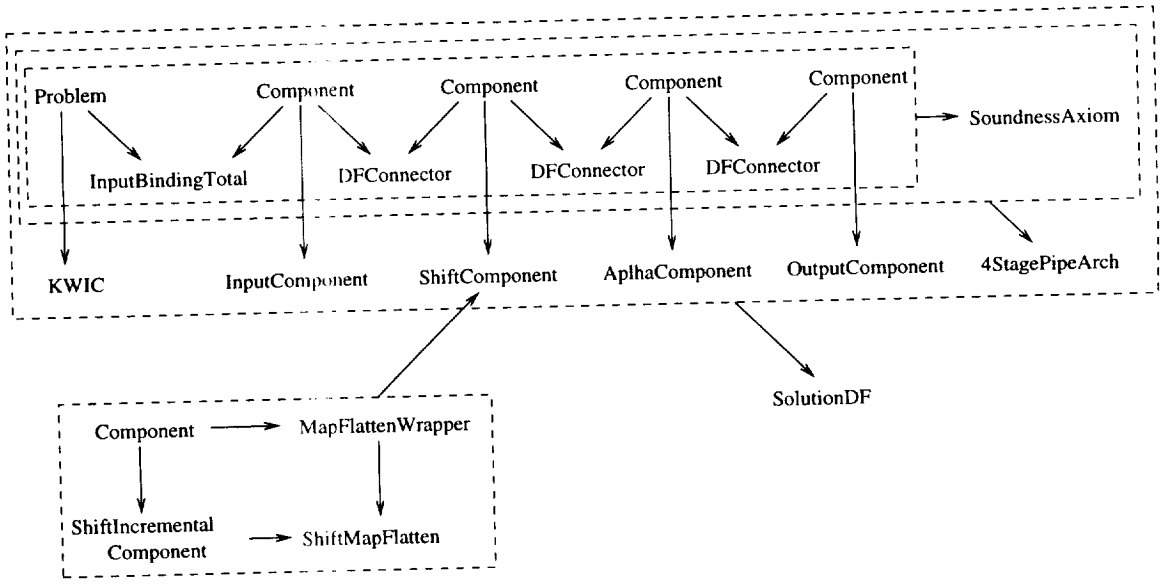

Figure 3: Specification Diagram for KWIC Data Flow Architecture

interface.

The second architecture we described was the reactive architecture, where components do not wait for the previous component to finish processing, but operate incrementally. For this example, it was necessary to specify an incremental version of the Alphabetize component. The original Alphabetize component could be built using an incremental version and a wrapper. However, this unnecessarily limits the sorting algorithm to insertion sort, pointing out a tradeoff between the data flow and reactive architectures: the performance gained by the concurrent execution of components in the reactive style results in a potential performance loss as a result of limiting the sorting algorithm.

This experience also pointed out the importance of wrappers in component integration. We believe that it may be useful to consider wrappers as first class entities in an architecture description language.

4 Related Work

This approach to architecture modeling arose out of an effort to model component adaptation tactics within the REBOUND framework [9, 10]. The architecture specification method (in it's monolithic form) was tested on the architecture of an AI-based control system for a deep-space probe [11].

Most efforts to formalize software architecture are targeted at formalizing styles and not with the problem decomposition aspects of architecture. Two approaches use algebraic theories to specify architectures. Marconi et. al. [8] use theory-based architecture representations to support architecture refinement. This work is concerned with architecture implementation and could be used to specify links between architecture specifications and architecture schemas. Gerken [5] also uses theories as the main unit of specification. We believe that the process logic descriptions of architectures are too operational to effectively model the relationships we are interested in.

5 Future Work

We are currently experimenting with a more general component model that will provide better support for concurrency and event-based communication. We are attempting to balance the expressibility of languages like Wright [2] (based on CSP) with the need to model requirements without implementation bias. We are also interested in mapping component and connector implementations onto commercial component platforms, such as JAVA beans. This could provide a reliable method for NASA to use COTS components in critical systems, by formally verifying a reference architecture for flight-critical applications built on top of a reliable platform.

References

[1] Gregory D. Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4), 1995.

[2] R. Allen and D. Garlan. Formalizing Architectural Connection. In *Proc. Sixteenth International Conference on Software Engineering*, pages 71–80, May 1994.

[3] Ted J. Biggerstaff and Alan J. Perlis, editors. *Software Reusability - Concepts and Models*, volume 1. ACM Press, 1989.

[4] R. M. Burstall and J. A. Goguen. Putting theories together to make specifications. In *IJCAI5*, pages 1045–58, 1977.

[5] Mark J. Gerkin. *Formal Foundations for the Specification of Software Architecture.* PhD thesis, Air Force Institute of Technology, March 1995.

[6] Richard Jüllig and Yellamraju V. Srinivas. Diagrams for software synthesis. In *The Eight Knowledge-Based Software Engineering Conference*, pages 10–19. IEEE, September 1993.

[7] Michael Lowry, Klaus Havelund, and John Penix. Verification and validation of AI systems that control deep-space spacecraft. In *Proceedings of the 10th International Symposium on Methodologies for Intelligent Systems (ISMIS'97)*, oct 1997. Invited Paper.

[8] Mark Moriconi, Xiaolei Qian, and Bob Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995.

[9] John Penix. *Automated Component Retrieval and Adaptation Using Formal Specifications.* PhD thesis, University of Cincinnati, April 1998.

[10] John Penix and Perry Alexander. Toward automated component adaptation. In *Proceedings of the Ninth International Conference on Software Engineering and Knowledge Engineering*, pages 535–542. Knowledge Systems Institute, June 1997.

[11] John Penix, Perry Alexander, and Klaus Havelund. Declarative specification of software architectures. In *Proceedings of the 12th International Automated Software Engineering Conference*, pages 201–209. IEEE Press, nov 1997.

[12] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.